 Bilkent University

CS 319 - Object-Oriented Software Engineering

Design Report

Project Group 11 - LUDO

Ahmet Taha Albayrak - 2131440

Alp Pehlivanoğlu - 21202023

Asli Cengiz – 21301183

Mustafa Motani 21402995

**TABLE OF CONTENTS**

**1.Introduction…………………………………………………………..………..…………………….……….…3**

**1.1. Purpose of the System…………………………….…………………………….……………3**

**1.2. Design Goals………………….………………..…………….……………………………………3**

**1.3. Definitions, Acronyms and Abbreviations……………………………………………4**

**1.4. References…………………….………………..…………….……………………………………4**

**2. Software Architecture…………………………..…………………....……………………………………5**

**2.1. Overview……………………………….…….…….………………….………………………..…5**

**2.2. Subsystem Decomposition…….…….…………………………………………………..…5**

**2.3. Architectural Styles…….…….……………………………………………………….…….…6**

**2.3.1. Layers…………………………………...….…………………………………………….…...…6**

**2.3.2. Model View Controller..….………………………………………………..………....…6**

**2.4. Hardware/Software Mapping..….………………………………………..…….…….…7**

**2.5. Persistent Data Management………….……………………...………………….………7**

**2.6. Access Control and Security……………….…….………………..……………….....….7**

**2.7. Boundary Conditions……………….…….……………………………….………………….7**

**3. Subsystem Services…………………….….……………………………………………….……………8**

**3.1. Design Pattern……………….…………….…………….…………….…………….……….…8**

**3.2. User Interface Subsystem…….………...………………………….…..…………….……9**

**3.3. Game Management Subsystem……………...…………...…………………...……..14**

**3.4 Game Entities Subsystem…...………………...………………....…..…………….……20**

**1. Introduction**

**1.1 Purpose of the System**

Ludo is a traditional board game of Southeast Asia. It is very easy to play multi-player strategy game whereby each player races his or her four tokens to the finish line. Movement of each player’s tokens is determined by die roll. However, in long run Ludo becomes mundane especially when there are only two players therefore we have included several other features to make it more addictive and fun to play for e.g. interactive and colorful user interface, added power ups and bonuses which makes Ludo more unpredictable and enjoyable.

**1.2 Design Goals [3]**

At this stage we have following goals to accomplish as our primary targets:

* **Easy to understand:** Even though the game itself is very easy to play on board, playing on computer can be difficult for beginners. We plan to keep it as simple as possible for the end users with on screen clicks to select the tokens that needs to be moved and highlighting the possible boxes that token can move. Moreover, we will have sound for alerts like killing a token of other players. Also we will have help window where players can read about how the program works with easy navigational windows
* **Adaptability:** We shall implement our game in JAVA environment primarily because we want to keep our program platform independent which we can’t get on C++ even though it has better performance. Moreover, we plan to use some built in JAVA libraries like ArrayList, JFrame in our program that makes it more suitable to use JAVA.
* **Efficiency vs Memory:** Since our program will not be so large memory is not our main concern. We aim to achieve best possible runtime of our program, so that player doesn’t have to wait a lot after moving a token. We will be not concern with memory much to get fastest possible response time from computer. Moreover, triggering a bonus should be immediately after token has moved. With slow response players might get annoyed and give up on game. Therefore, efficiency is our primary concern.
* **Extendibility and Modifiability:** We have divided our program into many smaller subsystems making it easy for us to change a particular section without affecting whole program. Even adding new features is easier as you just need to extend previously implemented classes.
* **Attractive Graphics:** We would include very attractive board designs and tokens to give traditional game a new look. Also triggering power ups will give nice animations related to that power up. Using innovative graphics can attract many users to the game.
* **Reliability:** By separating different subsystems we aim to keep potential unforeseen bugs doing minimum damage. Moreover, game will be saved after certain number of turns automatically in case of unexpected loss of game like power failure. Boundary conditions are taken into special consideration while implementing.

**1.3 Definitions, Acronyms and Abbreviations**

JRE: Java Runtime Environment

JDK: Java development Kit [2]

JVM: Java Virtual Machine [1]

**1.4 References**

[1] https://en.wikipedia.org/wiki/Java\_virtual\_machine

[2] <http://en.wikipedia.org/wiki/Java_(programming_language)>

[3] *Object-Oriented Software Engineering, Using UML, Patterns, and Java, 3rd Edition*, by Bernd Bruegge and Allen H. Dutoit, Prentice-Hall, 2010, ISBN-10: 0136066836.

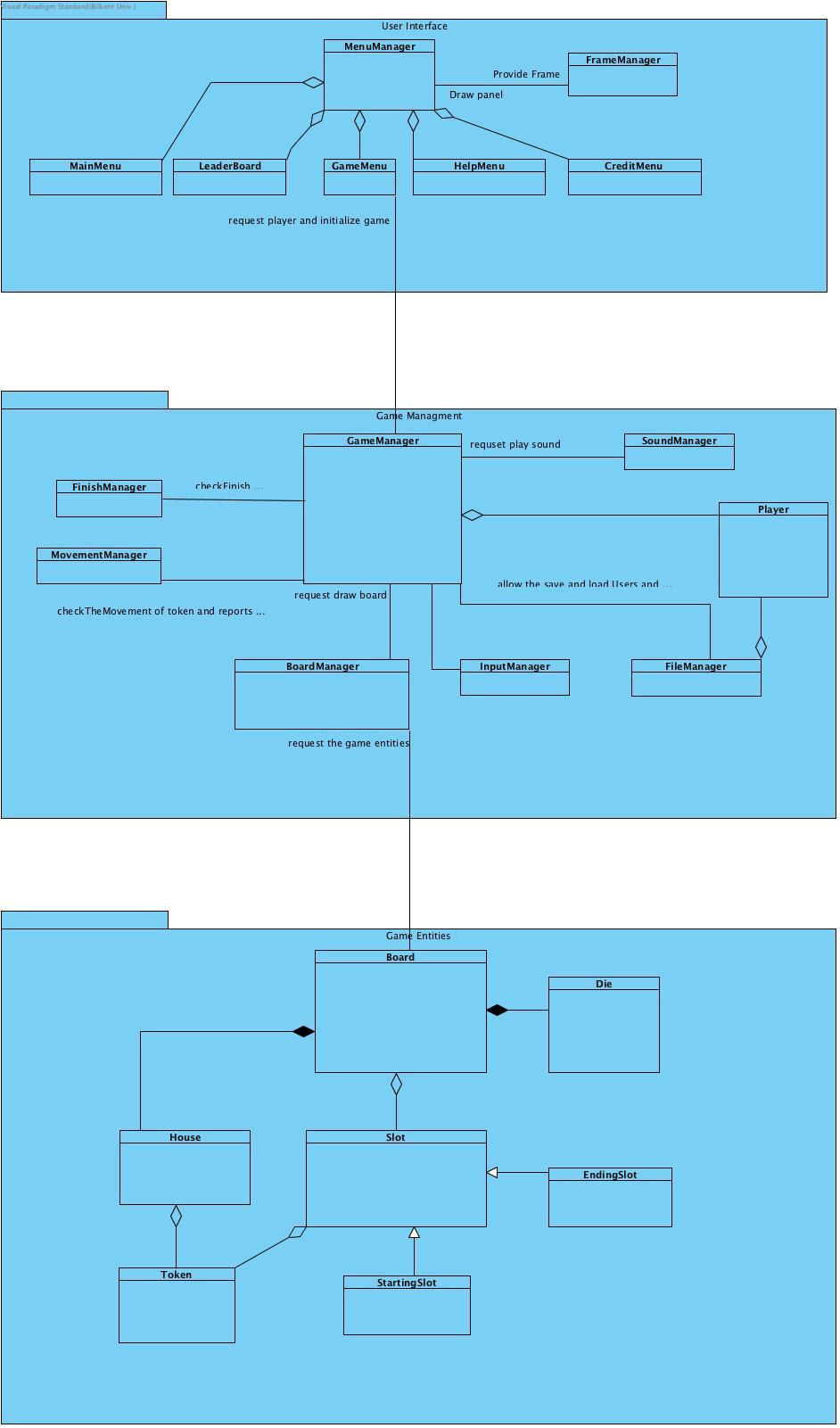
**2.Software Architecture**

**2.1 Overview**

In this section, decomposition our system into subsystem will be emphasized. Dividing into subsystem will reduce the coupling events between those subsystems. MVC is most fundamental architectural to differentiate model, view and controller classes. While decomposing the system MVC architectural style is used.

**2.2 Subsystem Decomposition**

Our system is decomposed into three subsystems which are called “User Interface”, “Game Management” and “Game Entities”. User Interface Subsystem includes classes that provide an interface for users to interact with the system. This subsystem handles the display of main menu and other screens. Game Management Subsystem includes classes that handle the events and user interactions, user choices. If user chooses to play the game, then GameManager creates and changes the game environment using the classes included and by interacting with the Board class in the Game Entities subsystem. Game Entities Subsystem includes classes that creates the objects which construct the game such as die, tokens and board. Board class in this subsystem puts together all the objects included in the game.



**2.3. Architectural Styles**

**2.3.1 Layers**

We have decomposed our system into three subsystems namely: User Interface, Game Management and Game Entities. Decomposition is based on the dependency of layers on each other. For instance, “User Interface” is independent of game objects therefore being at the top of hierarchy. Following “User Interface” is “Game Management”. Game Management contains the logic of the game for example movement of tokens, toggling player turns, rolling the dice, triggering bonuses etc. Last layer “Game Entities” contains the game objects for example tokens, board, dice etc. Our decomposition follows close architectural style therefore each layer can access layers below them as depicted in the Figure-1.

C:\Users\OzAtlangoc\AppData\Local\Microsoft\Windows\INetCacheContent.Word\figure.png

Figure-1

**2.3.2 Model View Controller**

In this architectural style, the main goal is creating three subsystem which are model classes, view classes and controller classes. In our system Game Entities layer contains the model classes. In MVC, controller classes should access the model classes. Game Management layer contains the control classes. As view part of MVC, this design has User Interface layer to interact with user. By this architecture it is achieved that interaction between user and system do not change the model of the system, therefore MVC is applicable to games.

**2.4. Hardware/Software Mapping**

Ludo will be implemented in Java programming language, so that it will require Java Runtime Environment in order to be executed. For the hardware requirements, keyboard and mouse inputs are necessary since with keyboard input user will type his/her name, and all of the movements and throwing the dice will be controlled by mouse input.

For the storage issue, we will consider to store high scores in .dat file. The game will be offline, and since it will be implemented with using java, it can work any desired system.

**2.5. Persistent Data Management**

Our game will store high scores in .dat file and that’s why it will not consume that much space.

**2.6. Access Control and Security**

In Ludo, we will consider to not hold any user information regarding to their private information. Since, our game will be offline, user’s system will not be affected negatively in terms of privacy by our game. Therefore, there will be no security problem in Ludo.

**2.7. Boundary Conditions**

**Initialization**

Since our game will be executable with .jar file, there will not be any installation in order to play the game.

**Termination**

Ludo can be terminated or closed by clicking “Quit Game” or “X” button from main menu screen. If user wants to terminate game during the game play screen, system provides termination chance through “X” button in order to perform quitting.

**Errors**

If any errors occur such as game manager could not be able to load sound and images, game will start without the sound and images that are not be loaded successfully.

**3. Subsystem Services**

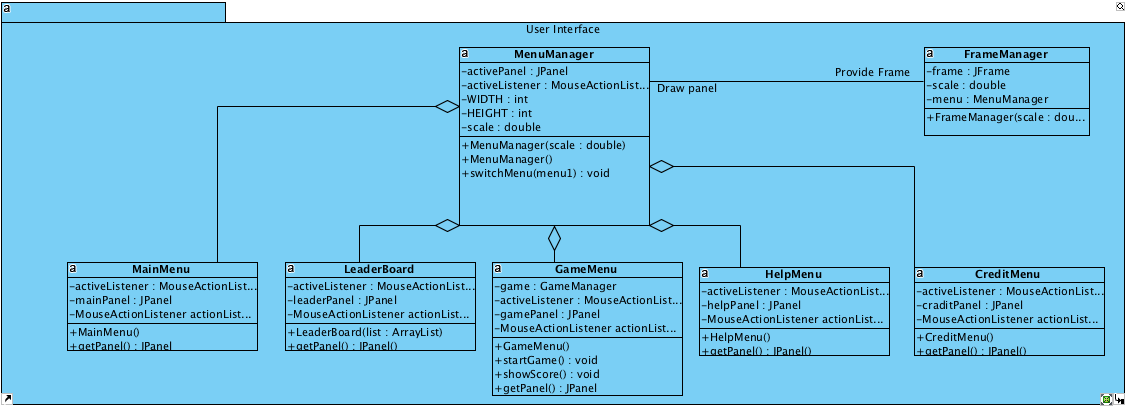
**3.1. Design Patterns**

**Façade Design Pattern**

Façade design pattern is a structural design pattern, which proposes that developers can easily manage a subsystem from a façade class since the communication between outside of this subsystem is performed only by this class. This pattern provides maintainability, reusability and extendibility since making changes in the façade class can reflect any change in the components of this subsystem.

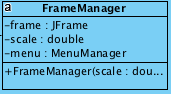
In our design we used façade pattern in all subsystems. In User Interface, façade class is MenuManager which able to create the menus and swap menus. In Game Management subsystem, façade class is GameManager, which communicates with the components of the Game Management subsystem according to the requests of User Interface subsystem. For Game Entities subsystem, façade class is Board class, which handles the operations on entity objects of our system, according to the needs of Game Management subsystem.

**3.2 User Interface Subsystem**



**Description:** User Interface subsystem provides the graphical drawing ability to system. This subsystem let the display different menu and manages the selection in those menus.

**Frame Manager Class**



**Attributes**

**private JFrame frame**: This is a frame in which displayed game objects.

**private double scale:** This is a scale for giving higher resolution display related to original game resolution.

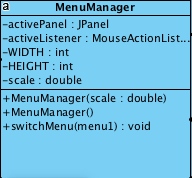
**private MenuManager menu**: This menu property allow the use MenuManager to create all menus. This is created and used while end of the program.

**Constructors**

**FrameManager(scale double)**: Initialize the *frame*, *scale* and *menu* properties.

**FrameManager() :** Initialize the *frame* and *menu* properties and set scale to 1.

**MenuManager**



**Attributes**

**private JPanel activePanel:** This JPanel type property of activePanel is used in graphical user interface to show actively used panel on screen.

**private final int WIDTH:** This int type property of WIDTH is used to determine maximum width of panel

**private final int HEIGHT:** This int type property of HEIGHT is used to determine maximum height of panel

**private double scale:** This double type property of scale is used to scale the panel sizes

**Constructors**

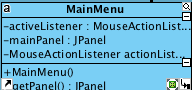
**MenuManager(scale : double) :** Initialize the *activePanel*, *scale* and *WIDTH* and *HEIGHT* properties.

**MenuManager() :** Initialize the *activePanel* and *WIDTH* and *HEIGHT* properties and set scale to 1.

**Methods**

**public void switchMenu(menu1):** Thismethod change the activePanel by input from user.

**MainMenu**



**Attributes**

**private MouseActionListener actionListener:** This attribute is to get the user input from the graphical user interface.

**private JPanel mainPanel:** This JPanel type attribute mainPanel is used in graphical user interface to show Main Menu on screen.

Constructors:

**MainMenu():**  Initialize the *mainPanel*, *actionListener* properties and graphical objects.

**Methods**

**public JPanel getPanel():** It return the JPanel type attribute mainPanel.

**LeaderBoard**



**Attributes**

**private MouseActionListener actionListener:** This attribute is to get the user input from the graphical user interface.

**private JPanel leaderPanel:** This JPanel type attribute leaderPanel is used in graphical user interface to show Leaderboard on screen.

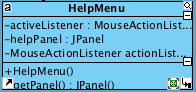
**Constructors**

**LeaderBoard():**  Initialize the *mainPanel*, *actionListener* properties and graphical objects.

**Methods**

**public JPanel getPanel():** It return the JPanel type attribute leaderPanel.

**HelpMenu**



**Attributes**

**private MouseActionListener actionListener:** This attribute is to get the user input from the graphical user interface.

**private JPanel helpPanel:** This JPanel type attribute helpPanel is used in graphical user interface to show Help Menu on screen.

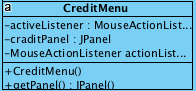
**Constructors**

**MainMenu():**  Initialize the *mainPanel*, *actionListener* properties and graphical object.

**Methods**

**public JPanel getPanel():** It return the JPanel type attribute helpPanel.

**CreditMenu**



**Attributes**

**private MouseActionListener actionListener:** This attribute is to get the user input from the graphical user interface.

**private JPanel creditPanel:** This JPanel type attribute creditPanel is used in graphical user interface to show Credits on screen.

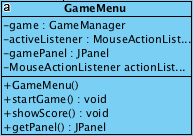
**Constructors**

**MainMenu():**  Initialize the *creditPanel*, *actionListener* properties and graphical object.

**Methods**

**public JPanel getPanel():** It return the JPanel type attribute creditPanel.

**GameMenu**



**Attributes**

**private JPanel gamePanel:** This JPanel type attribute creditPanel is used in graphical user interface to show Credits on screen.

**private game GameManager:** This GameManager type attribute game is used to create game and playing game.

**private MouseActionListener actionListener:** This attribute is to get the user input from the graphical user interface.

**Constructors**

**GameMenu():** Initialize the game*Panel* which will be able to let , *actionListener* properties and graphical object.

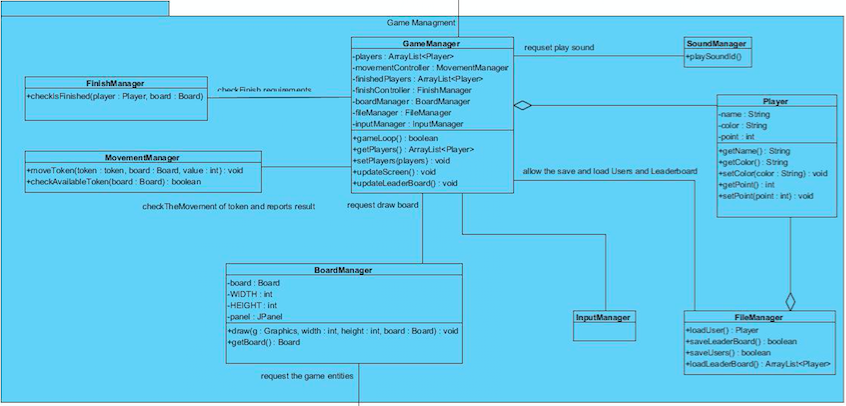
**Methods**

**public void startGame():** This method initialize the game and starts.

**public void showScore():** End of the game showScore clear the gamePanel and display the scores.

**public JPanel getPanel():** It return the JPanel type attribute gamePanel.

**3.3 Game Management Subsystem**



**GameManager**



**Description:** This class is accessed from game menu when player selects to start the game. This is the main game class and other manager classes are accessed from this class like SoundManager, FinishManager, MovementManager, BoardManager, FileManager and InputManager.

**Constructor**

**public GameManager()**: This constructor is invoked by game menu when player starts the game. It initializes all other managers.

**Attributes**

**private ArrayList<Player> players:** Contains all the players active in the game in an ArrayList.

**private ArrayList<Player> finishedPlayers:** It contains the players who have already finished the game.

**private MovementManager movementController:** It oversees the movement of tokens on the board.

**private FinishManager finishController:** It checks if the game is finished, or there are no more than 1 player still left in the game.

**private BoardManager boardManager:** It contains specifics of the board like dimensions and draws the board on the screen.

**private FileManager fileManager:** It contains the records of the high scores and players.

**private InputManager inputManager:** It is responsible for listening inputs from mouse and keyboard.

**Methods**

**public boolean gameLoop():** It checks the conditions to ensure that game hasn’t yet ended.

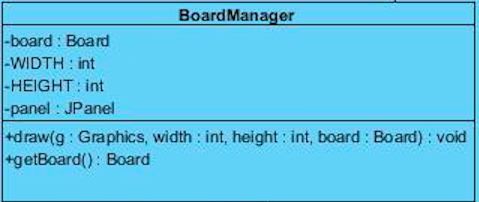
**public ArrayList<Players> getPlayers():** Returns the ArrayList of players in game.

**public void setPlayers(Player player):** Adds new player to the ArrayList.

**public void updateScreen():** Updates the screen after each move via boardManager object.

**public void updateLearderBoard():** At the end of the game it updates the leaderboard via fileManger object

**BoardManager**



**Description:** This class manages the contents of board and displays it on screen for players to interact with tokens.

**Constructor**

**public BoardManager():** Initiates the board drawing it on screen.

**Attributes**

**private Board board:** Contains all the objects that are shown on the board e.g. houses, tokens etc.

**private int width:** Width of the board.

**private int height:** Height of the board.

**private JPanel panel:** Panel for the contents of the board to be visible from.

**Methods**

**public void draw(Graphics g, int width, int height, Board board):** Draws the board on the screen using built in Graphics canvas.

**public Board getBoard():** Returns the instance of board.

**FinishManager**



**Description:** Ensures if game has finished or not.

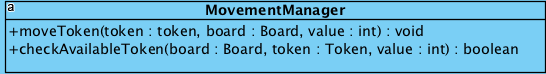
**Constructor**

**public FinishManager():** It contains checking mechanism to see if the game has finished or not.

**Method**

**public boolean checkIsFinished(Player player, Board board):** Checks for individual players on board if they have finished or not.

**MovementManager**



**Constructor**

**public MovementManager():** Initiates movementManager’s object that overlooks the movement of tokens on board.

**Method**

**public void moveToken(Token token, Board board, int value):** It moves the specified token on board by mentioned value.

**public boolean checkAvailableToken(Token token, Board board, int value):**  It checks if the specified token can move with particular value.

**SoundsManger**



**Description:** Whenever user eats another token it plays a unique sound.

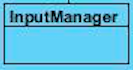
**Constructor**

**public SoundManager():** It holds the sounds that can be used on board.

**Methods**

**public void playSound():** It plays sound.

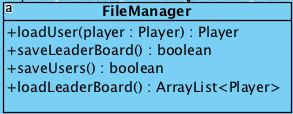
**InputManager**



**Description:**

It controls all the input that program receives from user. User will be able to move tokens via mouse clicks and enter name from keyboard.

**FileManager**



**Description:** It links to the file that contains records of leaderboard and all individual unique players.

**Constructor**

**public FileManager():** Initiates the FileManager object.

**Method**

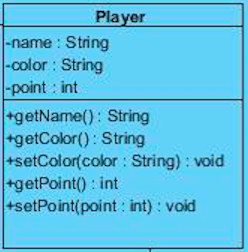
**public Player loadUser(Player p):** It looks for specific player in file and retrieves player’s information.

**public boolean saveLeaderBoard():** Saves leaderboard if it has been updated after an end of a game. Returns true if saving has been successful.

**public boolean saveUser(Player p):** Saves the player’s information in the file if it is unique. Returns true if done successfully.

**public ArrayList<Player> loadLeaderBoard():** Returns players on leaderboard in the form of ArrayList.

**Player**



**Description:** It holds all the attributes and methods related to individual player.

**Constructor:**

**public Player():** Initiates player object.

**Attributes:**

**private String name:** Stores name of the player.

**private String color:** Stores the color of the player on board.

**private int points:** Stores the points of that player so far in the game.

**Method:**

**public String getName():** Returns name of the player.

**public String getColor():** Returns color of the player.

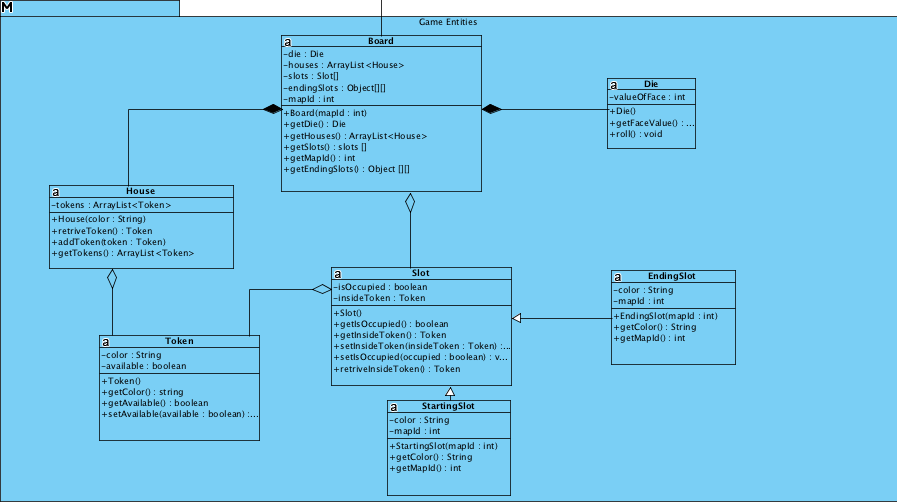
**public int getPoints():** Returns the points of the player.

**public void setName(String name):** Allows player to change his or her name.

**public void setColor(String color):** Select a new color.

**public int setPoints(int point):** Increase or decrease points.

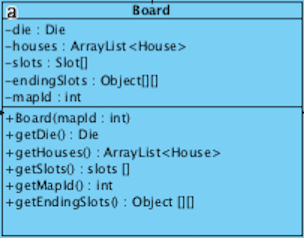
**3.4 Game Entities Subsystem**



**Description:** Game entitites subsystem consists each of the objects in the game. It has board class, which is a facade class, allows to access other classes. Board class has a relation with House, Slot and Die classes, and this relation shows the dependency. House and Slot classes have a relationship with Token class since Tokens have to either occupy a place in house or in a slot. Slot class inherits its methods and attributes to StartingSlot and EndingSlot classes.

**Board**

**Description:** Board class in one of the significant class in Ludo, since it includes all of the game objects that forms the game. From the real life, since board includes houses, slots, tokens and die, also in the code board class have a relationship with house and slot classes. That means, without board class, there is no game.



**Constructor**

**public Board(int mapId):** It constructs the Board according to map Id since map ID’s are important to specify the starting and the ending slots.

**Attributes**

**private Die die:** It holds the die object of the Die class.

**private ArrayList<House> house:** It holds the house objects of the House class in an array list.

**private Slot[] slots:** It holds the slot objects of the Slot class in an array.

**private Object[][] endingSlots:** In one dimension, it holds the integers and in the other dimension slot objects are held. Integers represent the players, slot objects represent the finishing slots of each player.

**private int mapId:** It holds the value of map ID in order to construct board.

**Methods**

**public Die getDie():** It allows to get the Die object in order to read Die values afterwards.

**public arrayList<Houses> getHouses():** It allows to get the house objects in an array list to construct the board.

**public int[] getSlots() :** It helps to get the values of slots in an array.

**House**

**Description:** House class creates house objects for each player and holds the tokens. For that reason, it has a relationship with Token class.



**Constructor**

**public House(String color) :** It constructs the house objects with determined token for each player. Players are represented by colors so color is the parameter of the constructor.

**Attributes**

**private ArrayList<Token> tokens:** It holds the token objects that are in the house.

**Methods**

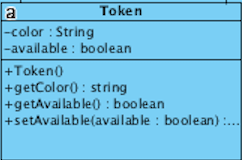
**public Token retrieveToken():** It helps to place tokens into slots.

**public addToken(Token token):** It allows to place tokens into houses.

**public arrayList<Token> getTokens:** It gets the existing tokens that are located in the houses.

**Token**

**Description:** All players have four tokens with a specific color that represents by each player.



**Constructor**

**public Token():** It creates token objects with colors that are determined by players.

**Attributes**

**private String color:** It holds the token colors that represent players

**private boolean available:** It holds whether the tokens are playable or not.

**Methods**

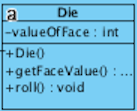
**public String getColor():** It gets the color for tokens.

**public boolean getAvailable():** It checks whether the tokens are playable or not.

**public void setAvailable(boolean available):** It makes tokens to playable for players.

**Die**

**Description:** Die class stands for creating the real die object in game. Player will play the token with respect to face value of dice.



**Constructor**

**public Die():** It constructs die object.

**Attributes**

**private int valueOfFace:** It has a random face value between 1-6.

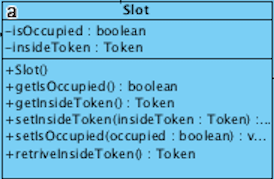
**Methods**

**public int getFaceValue():** This method provides the face value of the die.

**public void roll():** This method usesRandom class method in order to generate an integer value between one and six.

**Slot**

**Description:** It introduces the slots that tokens can be placed, and it helps to check whether the slots are occupied or not.



**Constructor**

**public Slot():** This constructor defines empty slots.

**public Slot(Token token):** It creates slots that are occupied by a specific token.

**Attributes**

**private boolean isOccupied:** It holds whether the slot object is occupied or not.

**private Token insideToken:** It is the specific token object that occupies a slot.

**Methods**

**public boolean getIsOccupied():** It gets the boolean value according to availability of specific slot.

**public Token getInsideToken():** If a specific slot is occupied by a token, it gets player to whom belong that token.

**public void setInsideToken(Token insideToken):** When a token is replaced by another token in a specific slot, it replaces the information of the old token with the new one.

**public void setIsOccupied(boolean occupied):** When a token is placed to slot, it sets that slot as occupied.

**public Token retrieveInsideToken():** When a token that occupied a slot, is to be moved in another slot, it will return that token and it sets the occupied slot as non-occupied.

**StartingSlot**

**Description:** When the player gets dice value of six, and wants to enter a new token into the game from the house, there is a specific slot for each player to place his/her token. This class defines those slots.



**Constructor**

**public StartingSlot(int mapId):** It constructs starting slot according to mapID.

**Attributes**

**private String color:** Since each player has specific starting slots, those slots are identified by these colors.

**private int mapId:** Since there are different maps with different starting slots exist, the information of mapId allows to specify these starting slots.

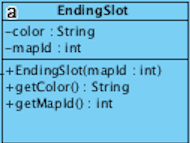
**Methods**

**public String getColor():** It gets the color of the player.

**public int getMapId():** It gets the mapId.

**EndingSlot**

**Description:** For players to finish the game, they have to place their tokens into a column that is specified for them. This class helps to specify the slot in each of those columns.



**Constructor**

**public EndingSlot(int mapId):** It constructs ending slots according to mapID.

**Attributes**

**private String color:** Since each player has specific ending slots, those slots are identified by these colors.

**private int mapId:** Since there are different maps with different ending slots exist, the information of mapId allows to specify these ending slots.

**Methods**

**public String getColor():** It gets the color of the player.

**public int getMapId():** It gets the mapId.